Northeastern University - Seattle

Khoury College of Computer Sciences

CS6200 Information Retrieval Fall 2019

Lecture 6: Query Processing

Oct 7, 2019

Administrivia

- Check corrected stuff every week!
  - Thank your TA whenever you see him!

- Reminder: No extensions possible for quizzes. Doesn't make sense to extend a deadline after solutions published/discussed.

- We'll look at Quiz 5 & Assignment 2 at the end of today's lecture.

Quiz 4 Discussion?

1. Recency, Source of news, Resolution, B&W vs. color, #clicks, Product ratings, …, location (news, tax site), Search history, Location… [Not product features, but document/site features]
2. One solution: Take deltas, count digits. 100 bytes vs 46; to compress till further, sort them first and then take deltas; 39 bytes. NEXT
3. Zero and negative integers. Usual ways, e.g. add a big number, or map to odd & even numbers
4. <Reduce description> The shuffle step ensures that the right data gets to the right Reducer, using e.g. hashing.
4. Always show top 10? No. Site diversity. Topic diversity. Diff. doc types. Location. Timeliness. “Discovery”

Answer to Quiz 4, question 2

Number	Last 4	Deltas	Sorted Last 4	Deltas after sort
2065553392	3392		1102	
2065554610	4610	1218	2272	1170
2065559209	9209	4599	3392	1120
2065556059	6059	-3150	4085	693
2065555036	5036	-1023	4610	525
2065552272	2272	-2764	5036	426
2065554085	4085	1813	5453	417
2065555453	5453	1368	6059	606
2065551102	1102	-4351	6096	37
2065556096	6096	4994	9209	3113
10 * 10 = **100** bytes		10 + 9*4 = **46** bytes		10 + 29 = **39** bytes

I saw some other interesting approaches. Generally anything that compresses is a good answer. But some, like Elias delta etc., are over-kill. I'd indicated these are phone numbers, and to look at digits (not bits). Also, I should have said you can't ignore the first six digits.

Also, even if I don't explicitly ask you, please always explain your answers. Thanks.

Oct 14 lecture

- Options
  - Reschedule for on-ground lecture. [preferred, but at what time? Tuesday Oct 15th?]
  - I record the lecture and release the audio and the slides.

Overview

- Query Processing
  - Document at a time
  - Term at a time
- Optimization
- Conjunctive Queries
- Threshold Methods, Structured Queries
- Distributed Evaluation
- Caching
- PageRank [linked to Lecture #4]

QUERY PROCESSING

Why Query Processing?

We have a query, and we have an index

So: Why do we need query processing?

Optimize processing,
so we reduce
time to return results

Query Processing

Document-at-a-time

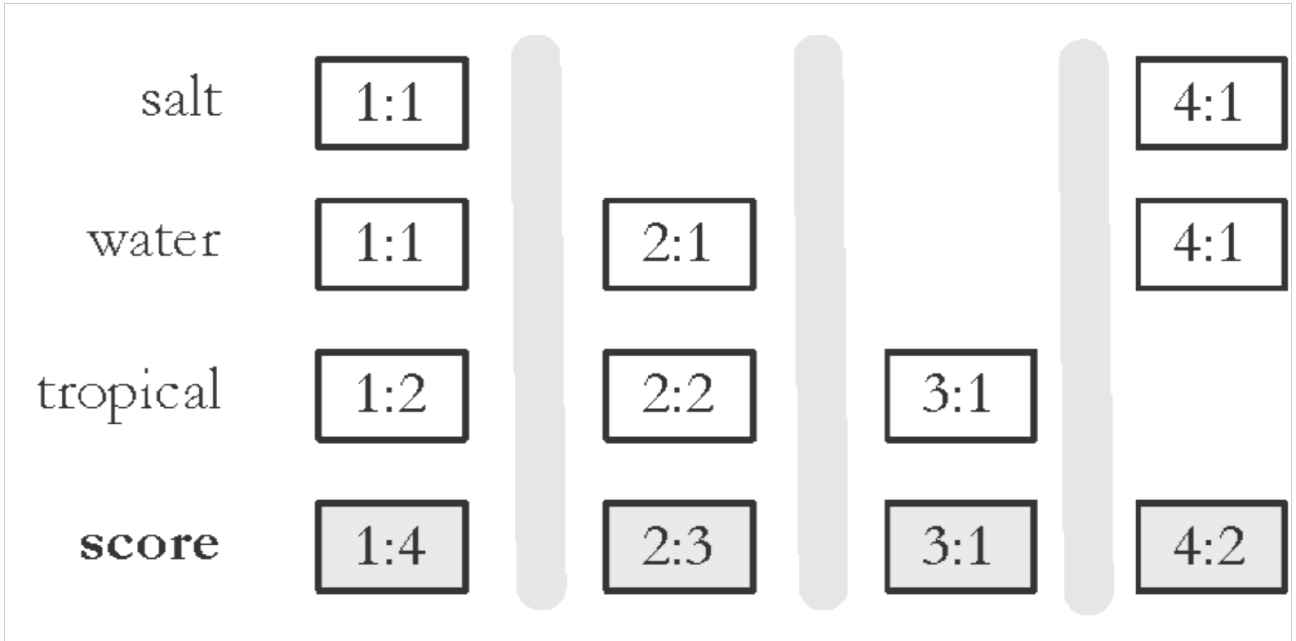
Calculates complete scores for documents by processing all term lists, one document at a time

Term-at-a-time

Accumulates scores for documents by processing term lists one at a time

Both approaches use optimization techniques that significantly reduce time to generate scores

Document-At-A-Time

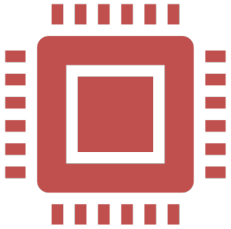


Assume score is just the total of the counts of the query terms in a document.

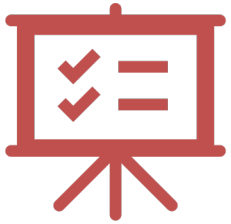
Document-At-A-Time

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all documents d ∈ I do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) f_i(l_i)          ▷ Update the document score
            end if
            l_i.movePastDocument( d )
        end for
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

Improving Document-at-a-time



Very low memory (accumulator) requirement

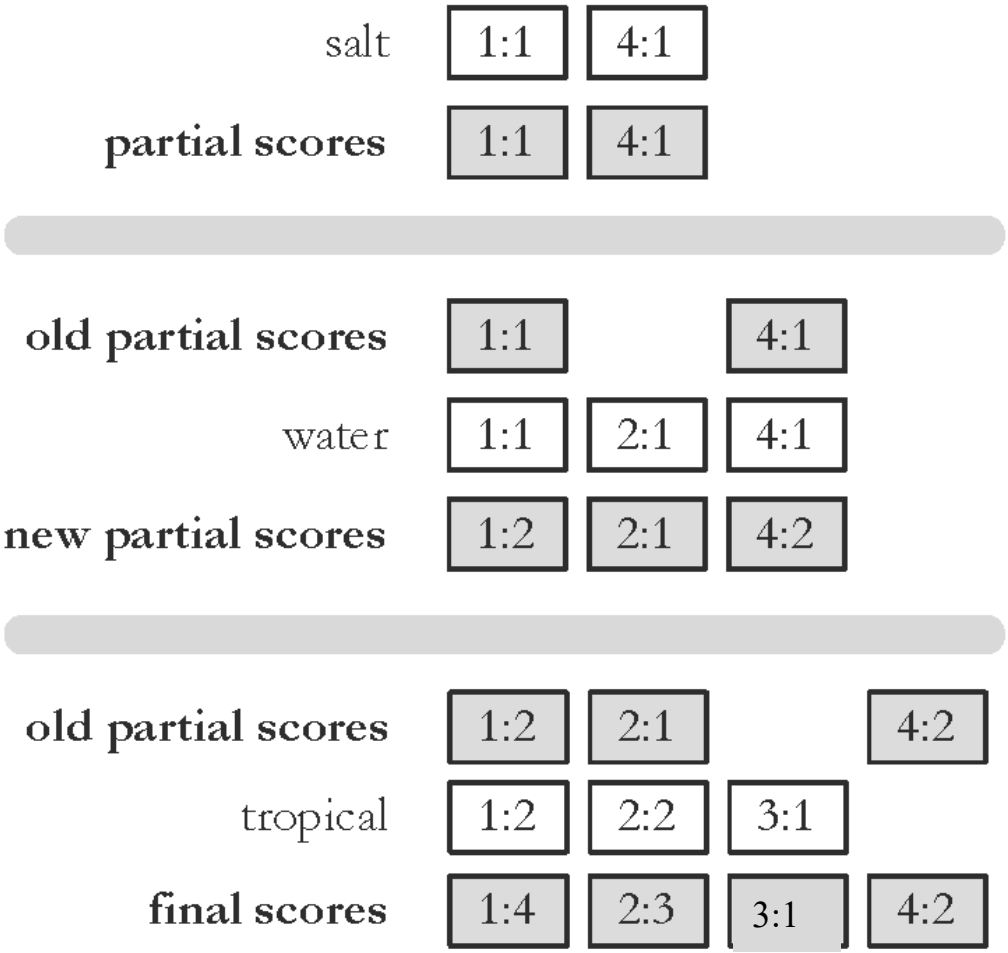


But can do we improve on this?

How many result documents do we need to keep track of?

Do we need to scan all documents?

Term-At-A-Time



Term-At-A-Time

```
procedure TermAtATimeRetrieval(Q, I, f, g k)
    A ← HashTable()
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all lists l_i ∈ L do
        while l_i is not finished do
            d ← l_i.getCurrentDocument()
            A_d ← A_d + g_i(Q)f(l_i)
            l_i.moveToNextDocument()
        end while
    end for
    for all accumulators A_d in A do
        s_d ← A_d                    ▷ Accumulator contains the document score
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

Term-at-a-time

SCORES ARE THE SAME,

USES MORE MEMORY FOR ACCUMULATORS,

BUT ACCESSES DISK MORE EFFICIENTLY

Optimization Techniques

Broadly, two classes of optimization

Read less data from inverted lists

e.g. skip lists

better for simple feature functions

Calculate scores for fewer documents

e.g., conjunctive processing

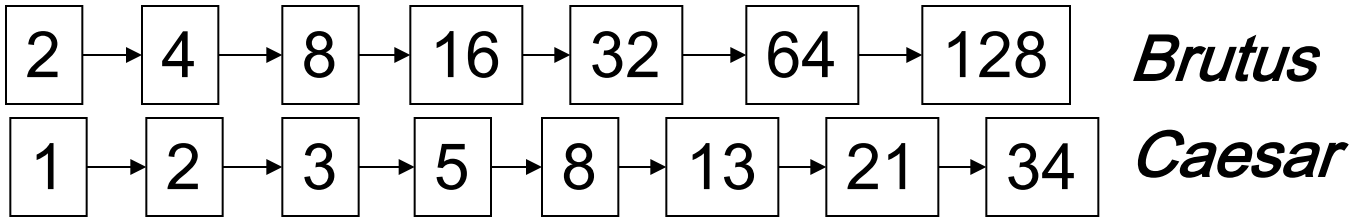
better for complex feature functions

Query processing: AND

- Result **must** contain all query terms ('AND')
- Consider processing the query:

  Brutus *AND* ***Caesar***

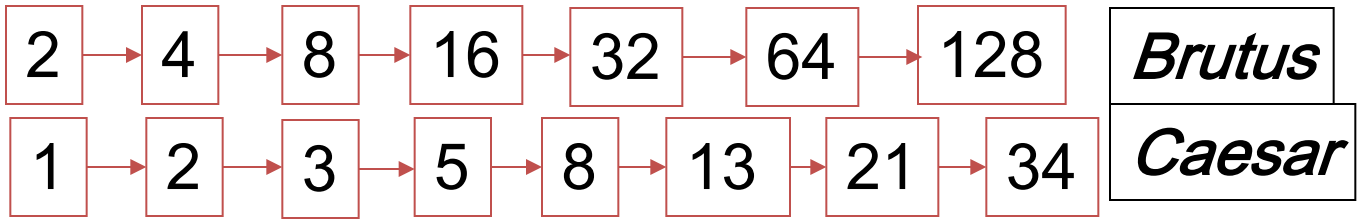
  – Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.
  – Locate ***Caesar*** in the Dictionary;
    - Retrieve its postings.
  – "Merge" the two postings (intersect the document sets):



CONJUNCTIVE QUERIES ('AND')

The merge

Walk through the two postings simultaneously, in time linear in the total number of postings entries



If the list lengths are *x* and *y*,
the merge takes O(x+y) operations.

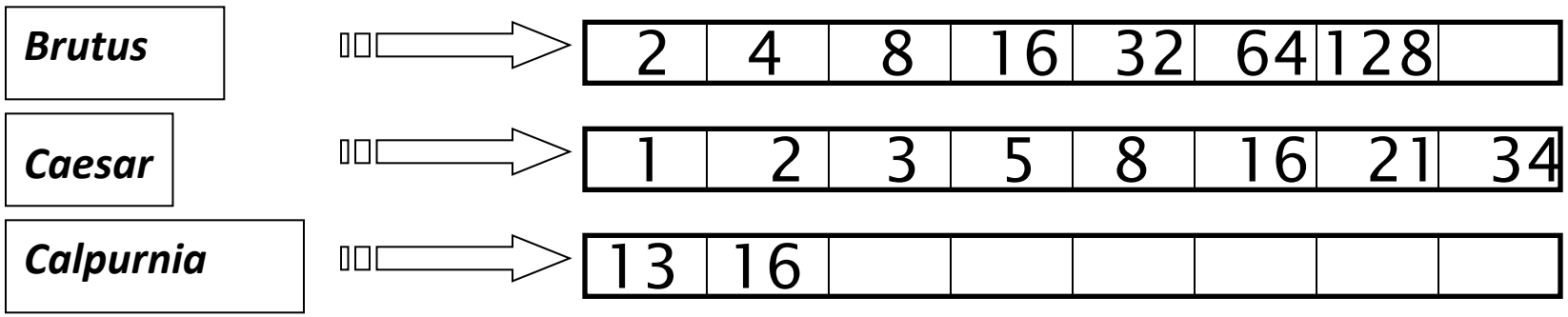
Crucial: postings sorted by docID

Intersecting two postings lists
(a "merge" algorithm)

```
INTERSECT(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then ADD(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8                then p1 ← next(p1)
 9                else p2 ← next(p2)
10  return answer
```

Query optimization

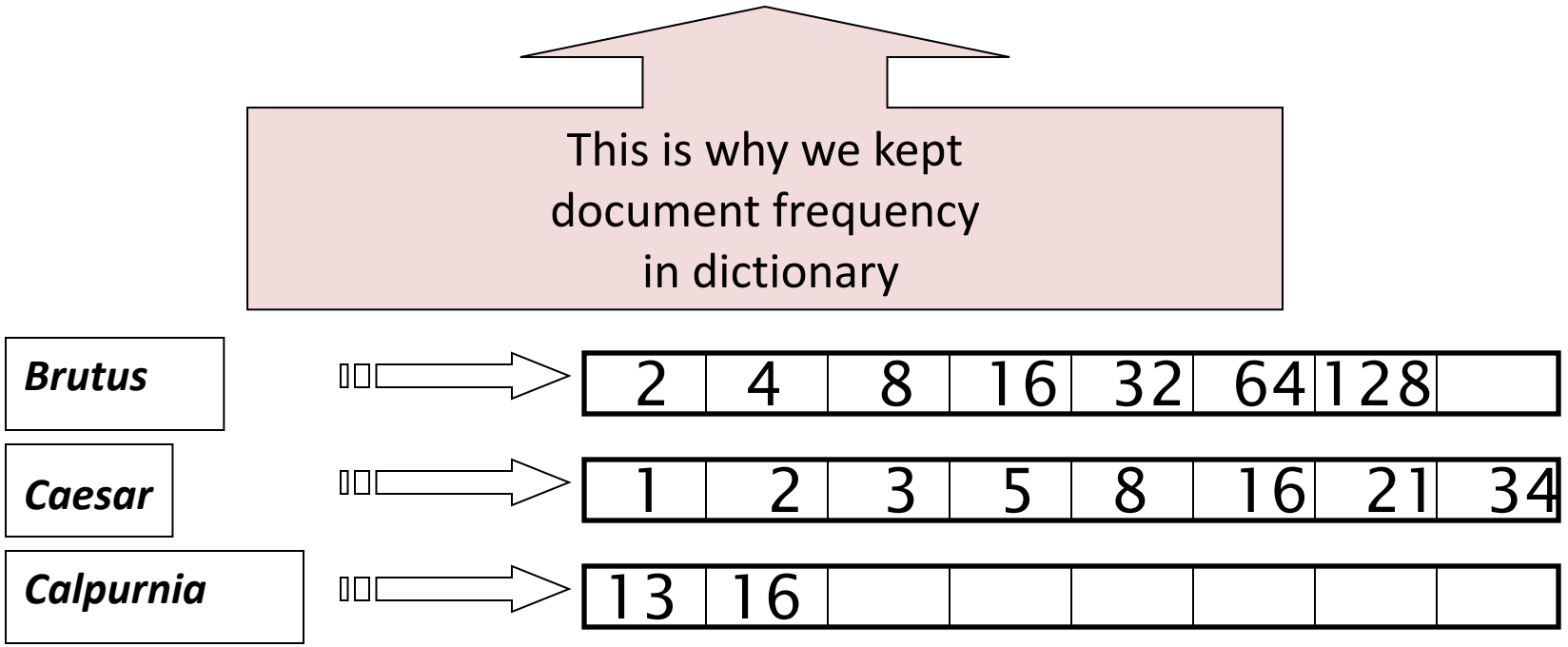
- What is the best order for query processing?
- Consider a query that is an *AND* of *n* terms.
- For each of the *n* terms, get that term's postings, then *AND* them together.



Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

Query optimization example

- Process in order of increasing frequency:
  - *start with smallest set, then keep cutting further*.

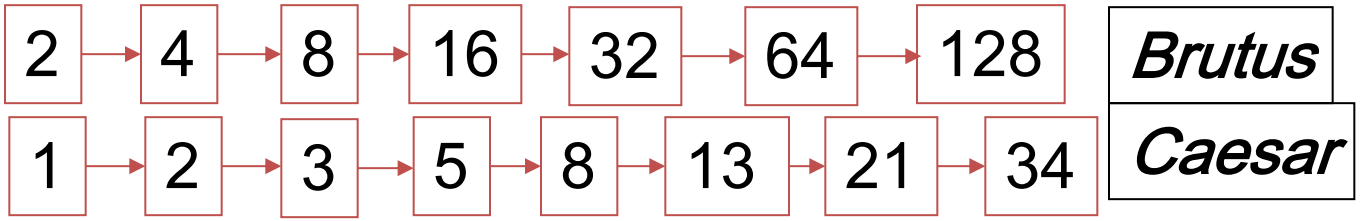


Execute the query as (***Calpurnia*** *AND* ***Brutus)*** *AND* ***Caesar***.

More general optimization

*Query: (**madding** OR **crowd**) AND (**ignoble** OR **strife**)*

- Get document frequencies for all terms.



- Estimate the size of each *OR* by …
  - the sum of its document frequencies (conservative)
- Process in increasing order of *OR* sizes

Conjunctive Term-at-a-Time

```
1: procedure TermAtATimeRetrieval(Q, I, f, g, k)
2:     A ← Map()
3:     L ← Array()
4:     R ← PriorityQueue(k)
5:     for all terms w_i in Q do
6:         l_i ← InvertedList(w_i, I)
7:         L.add( l_i )
8:     end for
9:     for all lists l_i ∈ L do
10:        d_0 ← −1
11:        while l_i is not finished do
12:            if i = 0 then
13:                d ← l_i.getCurrentDocument()
14:                A_d ← A_d + g_i(Q) f(l_i)
15:                l_i.moveToNextDocument()
16:            else
17:                d ← l_i.getCurrentDocument()
18:                d′ ← A.getNextAccumulator(d)
19:                A.removeAccumulatorsBetween(d_0, d′)
20:                if d = d′ then
21:                    A_d ← A_d + g_i(Q) f(l_i)
22:                    l_i.moveToNextDocument()
23:                else
24:                    l_i.skipForwardToDocument(d′)
25:                end if
26:                d_0 ← d′
27:            end if
28:        end while
29:    end for
30:    for all accumulators A_d in A do
31:        s_d ← A_d                ▷ Accumulator contains the document score
32:        R.add( s_d, d )
33:    end for
34:    return the top k results from R
35: end procedure
```

Processing first term

Checks for next accumulator entry containing all prev terms, and check for posting for that d’

Conjunctive Document-at-a-Time

```
1:  procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
2:      L ← Array()
3:      R ← PriorityQueue(k)
4:      for all terms w_i in Q do
5:          l_i ← InvertedList(w_i, I)
6:          L.add( l_i )
7:      end for
8:      d ← −1
9:      while all lists in L are not finished do
10:         s_d ← 0
11:         for all inverted lists l_i in L do
12:             if l_i.getCurrentDocument() > d then
13:                 d ← l_i.getCurrentDocument()
14:             end if
15:         end for
16:         for all inverted lists l_i in L do
17:             l_i.skipForwardToDocument(d)
18:             if l_i.getCurrentDocument() = d then
19:                 s_d ← s_d + g_i(Q) f_i(l_i)        ▷ Update the document score
20:                 l_i.movePastDocument( d )
21:             else
22:                 d ← −1
23:                 break
24:             end if
25:         end for
26:         if d > −1 then R.add( s_d, d )
27:         end if
28:     end while
29:     return the top k results from R
30: end procedure
```

Find largest doc d

Try to skip all lists to doc d.
If possible, good; else choose another d

Threshold Methods .. 1

Threshold methods use number of top-ranked documents needed (*k*) to optimize query processing

for most applications, *k* is small

For any query, there is a *minimum score* that each document needs to reach before it can be shown to the user

score of the *k*th-highest scoring document gives *threshold* τ

optimization methods estimate τ' to ignore documents

Threshold Methods .. 2

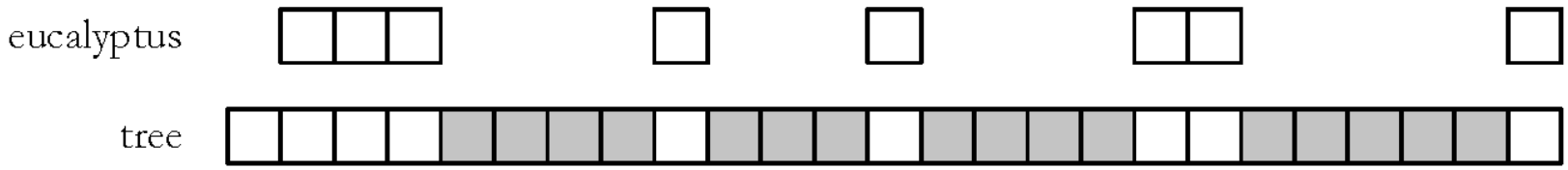
For document-at-a-time processing, use score of lowest-ranked document so far for τ'

for term-at-a-time, must use *kth*-largest score in the accumulator table

MaxScore method compares the maximum score that remaining documents could have to τ'

safe optimization in that ranking will be the same without optimization

MaxScore Example



- Indexer computes μ_{tree}
  - maximum score for any document containing just "tree"
- Assume k =3, τ' is lowest score after first three docs
- Likely that $\tau' > \mu_{tree}$
  - τ' is the score of a document that contains both query terms
- Can safely skip over all gray postings

Other Approaches

Early termination of query processing

- ignore high-frequency word lists in term-at-a-time,
- ignore documents at end of lists in doc-at-a-time …

→ *unsafe* optimizations

List ordering

- order inverted lists by quality metric (e.g., PageRank) or by partial score
- makes unsafe (and fast) optimizations more likely to produce good documents

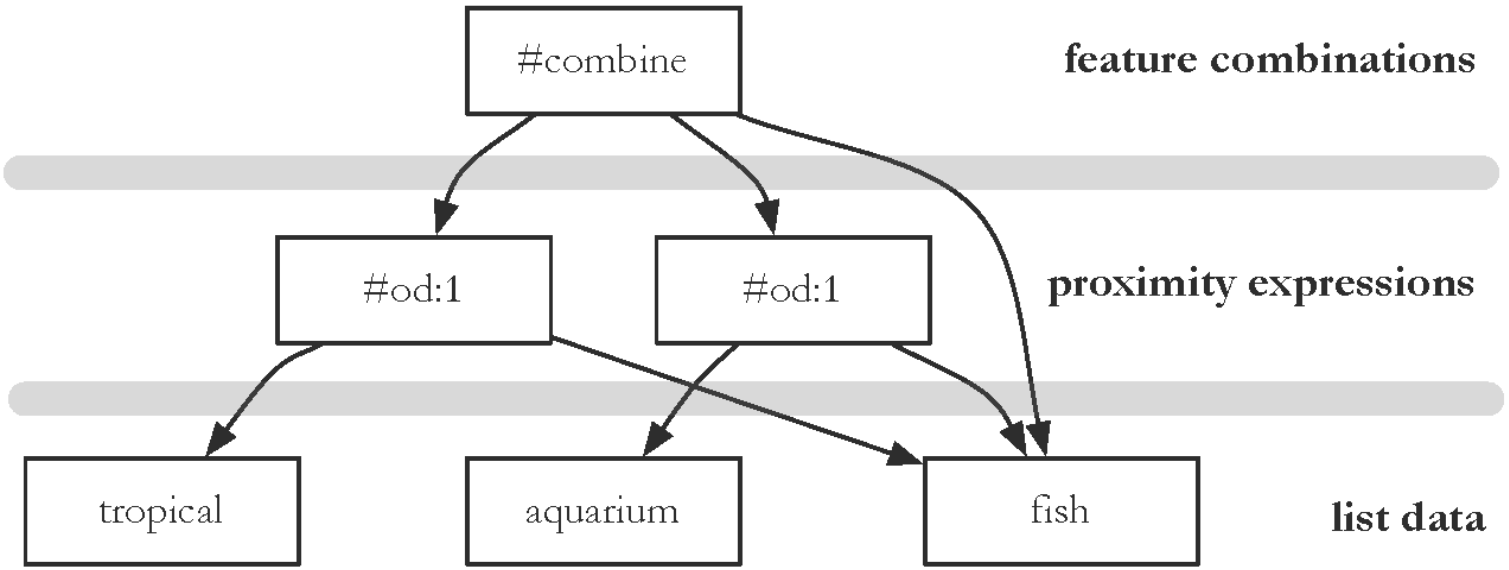
Structured Queries

Query language can support specification of complex features

- Like SQL for database systems
- *Query translator* converts the user's input into the structured query representation
- Galago query language is the example used here (CIIR, UMass and LTI, CMU)
- e.g., Galago query:

```
#combine(#od:1(tropical fish) #od:1(aquarium fish) fish)
```

Evaluation Tree for Structured Query



Distributed Evaluation

Basic process

- All queries sent to a *director machine*
- Director then sends messages to many *index servers*
- Each index server does some portion of the query processing
- Director organizes the results and returns them to the user

Two main approaches

- Document distribution
  - by far the most popular
- Term distribution

Distributed Evaluation

Document distribution

- each index server acts as a search engine for a small fraction of the total collection
- director sends a copy of the query to each of the index servers, each of which returns the top-*k* results
- results are merged into a single ranked list by the director

Collection statistics should be shared for effective ranking

Distributed Evaluation

Term distribution

- Single index is built for the whole cluster of machines
- Each inverted list in that index is then assigned to one index server
  - in most cases the data to process a query is not stored on a single machine
- One of the index servers is chosen to process the query
  - usually the one holding the longest inverted list
- Other index servers send information to that server
- Final results sent to director

Failures & Delays in Distributed Systems

- In a distributed system, what should happen if one server fails to respond, or is delayed?

- Timeouts

Caching

- Query distributions similar to Zipf
  - About half the queries each day are unique, but some are very popular
  - 9/11 situation; spelling
- Caching can significantly improve effectiveness
  - Cache popular query results
  - Cache common inverted lists
- Inverted list caching can help with unique queries
- Cache must be refreshed to prevent stale data
  - At restart, need to “warm up” caches. How?

TIME FOR A QUICK STRETCH !

RANDOM SURFERS & PAGERANK

CONTEXT: DOCUMENT PROCESSING & LINK EXTRACTION

Links and PageRank

Why extract links?

- Anchor Text
  - Assumption: Anchor text describes content of target page
- Link Graph
  - Considered important ranking factor. So search goes like this:
    - Look for words in query
    - Look for highest quality page
    - Return highest-quality-page that has-query-words
- Assumptions:
  - Links denote connections to important or quality sites
  - More inlinks ➔ more important? high quality?

Origins of PageRank

- Citation Analysis
  - Dumais et al [1999] suggest that….
- Not just #citations, but who cites you (which article from which journal etc.)
- Lots of work by **Eugene Garfield,**
  Institute for Scientific Information, Philadelphia
  - Science Citation Index: who cited whom
- Uses:
  - Article impact, Academic impact & promotions!
  - Article similarity (overlap of other article citing both)
  - Emerging areas (papers citing each other)

Citations → Hyperlinks

- Citations in academic literature like hyperlinks

→ PageRank [Page et al, 1999] for computing quality of web pages

Lawrence Page, Sergey Brin, Rajeev Motwani & Terry Winograd, (1999)
The PageRank Citation Ranking: Bringing Order to the Web.
Stanford InfoLab Technical Report.

Model behind PageRank

Imagine a web surfer surfing randomly

Start at a random page

At each step, click on any one of the links on the page (with equal probability)

Each page has a long-term visit rate (in steady state)

That is the page's 'PageRank'

Random Surfer Model

Browse the Web using the following algorithm:

Choose a random number r between 0 and 1

If $r < \lambda$: Jump to a **random page**

If $r \geq \lambda$: Click **a random link** on current page

Start again

Will visit all pages

PageRank of a page is the probability that the "random surfer" will be looking at that page

links from popular pages will increase PageRank of pages they point to

Dangling Links

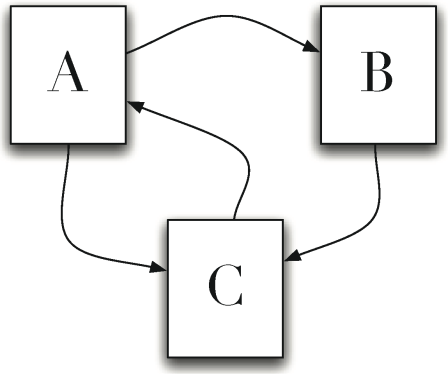
Random page jump prevents getting stuck on pages that:

- do not have links
- contains only links that no longer point to other pages
- have links forming a loop

Links that point to the first two types of pages are called *dangling links*

- may also be links to pages that have not yet been crawled

PageRank … 1



PageRank (*PR*) of page C = *PR*(A)/2 + *PR*(B)/1

More generally,

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L_v}$$

where B_u is the set of pages that point to u, and L_v is the number of outgoing links from page v (not counting duplicate links)

PageRank … 2

Don’t know PageRank values at start

Assume equal values (1/3 in this case), then iterate:

- first iteration: *PR*(C) = 0.33/2 + 0.33 = 0.5, *PR*(A) = 0.33, and *PR*(B) = 0.17
- second: *PR*(C) = 0.33/2 + 0.17 = 0.33, *PR*(A) = 0.5, *PR*(B) = 0.17
- third: *PR*(C) = 0.42, *PR*(A) = 0.33, *PR*(B) = 0.25

Converges to *PR*(C) = 0.4, *PR*(A) = 0.4, and *PR*(B) = 0.2

PageRank … 3

- Taking random page jump into account,
  1/3 chance of going to any page when $r < \lambda$
- $PR(C) = \lambda/3 + (1 - \lambda) \cdot (PR(A)/2 + PR(B)/1)$
- More generally,

$$PR(u) = \frac{\lambda}{N} + (1 - \lambda). \sum_{v \in B_u} \frac{PR(v)}{L_v}$$

  - where N is the number of pages, and λ typically 0.15

```
1: procedure PageRank(G)
2:     ▷ G is the web graph, consisting of vertices (pages) and edges (links).
3:     (P, L) ← G                                  ▷ Split graph into pages and links
4:     I ← a vector of length |P|                  ▷ The current PageRank estimate
5:     R ← a vector of length |P|   ▷ The resulting better PageRank estimate
6:     for all entries I_i ∈ I do
7:         I_i ← 1/|P|                 ▷ Start with each page being equally likely
8:     end for
9:     while R has not converged do
10:        for all entries R_i ∈ R do
11:            R_i ← λ/|P| ▷ Each page has a λ/|P| chance of random selection
12:        end for
13:        for all pages p ∈ P do
14:            Q ← the set of pages    such that (p, q) ∈ L and q ∈ P
15:            if |Q| > 0 then
16:                for all pages q ∈ Q do
17:                    R_q ← R_q + (1 − λ)I_p/|Q|       ▷ Probability I_p of being at
   page p
18:                end for
19:            else
20:                for all pages q ∈ P do
21:                    R_q ← R_q + (1 − λ)I_p/|P|
22:                end for
23:            end if
24:            I ← R                       ▷ Update our current PageRank estimate
25:        end for
26:    end while
27:    return R
28: end procedure
```

Issues with PageRank

Real surfers not truly random surfers

bookmarks, jumps via other searches, back buttons

Markov model not a good model of surfing

Bad results from PageRank

Consider query : [Microsoft]

- Microsoft End User License Agreement has many inlinks from many good Microsoft pages, so high PageRank
- contains the word Microsoft

But EULA page not a great result for this query

Is PageRank important?

Claimed to be most important part of at least one search engine's ranking

Truth:

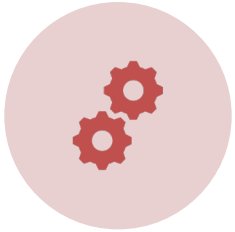
many components important: anchor text, proximity, phrases…

known issues : EULA

still important factor

link spam still an issue (cf. Google Bombing, e.g. "Miserable Failure")

Summary



Looked at various query processing methods, and ways to optimize them

Looked briefly at distributed evaluation

PageRank

Next week: **Query Refinement**

Readings

Croft, Metzler & Strohman (CMS):
Sections 5.7 (Query Processing), 4.5 (PageRank)

Any article covering Google/Bing advanced operators, such as https://bynd.com/news-ideas/google-advanced-search-comprehensive-list-google-search-operators/

Optional: Page, Lawrence and Brin, Sergey and Motwani, Rajeev and Winograd, Terry (1999) *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report. Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

Optional: Manning, Raghavan & Schütze (MRS), Sections 1.3, 2.3, Chap 21 except sections 21.2.2, 21.3

Quiz 5

Assignment 2

Questions?